Current Technologies in Automatic Test Suites Generation and Verification of Complex Systems

Darabant Sergiu Adrian -Loria Nancy

October 24, 1999

1 Introduction

We have in our days more complex systems than ever in the so called *information age*. Everything is based on the ability to manage information, more information each moment. The success of this aim is based on the communication systems which are, in turn, more developed and, as a consequence, more complicated. In order to control this complexity explosion new architectures have been developed, architectures that hide details that are not relevant for a real system, implemented above all these. This alleviates the need to concentrate on all the general, non important aspects; for example the communication layer in a data exchanging architecture. It remains for the developer to concentrate on data exchange issues and not on the communication ones. Open systems are another example of this. Here multivendor components operate in a meaningful way in order to achieve a common goal.

Testing and verifying such complex systems where many details are hidden becomes an important issue, especially if they are meant to co-operate. It is not a handy task to manually test and prove correctness requirements for a complex communicating system or software system. Proving that an implementation is correct is feasible only for very small entities. Testing systems, on the other side, requires them to be formally specified. The task of manually generating test sequences is acceptable also only for simple systems but not feasible for complex ones. This is due, here, to the large number of cases that should be considered. Therefore, a more practical approach is to test every manufacturer's system against a single test source, that corresponds to the system standard specification. This is generally known as *conformance testing* - "the process of verifying that an implementation is in accordance with a particular standard"[Kni93].

1.1 Importance of formal methods and testing

A well formed specification is a crucial point in software development, and especially in the development of complex systems such as protocols, systems that model the complex outside world by means of object orientation, etc. Hardware tends to grow in scale as we are approaching the speed of light and nothing seams to be left to do, other than growing in scale or multiplying in dimensions. It is likely that it will be more easily for subtle errors to appear because of this fact. Increase in complexity leaves place for errors. As some of these errors may cause catastrophic losses, one of the most challenging tasks in software system design is to assure reliability.

Software maintenance is another primary concern, as statistics show that much of the system's costs stem from testing and maintenance stages of the software lifecycle process. A

significant portion of this cost is due to the absence of rigorous practices that eliminate residual specification and design errors which causes imprecision, ambiguity and errors.

Using formal methods has as result an added reliability. They provide a notation for the formal specification of the system whereby the desired properties are described in a way that can be reasoned about formally, in mathematics, or informally. The use of formal methods does not guarantee correctness but offers the advantages of mathematical analysis of software design using a mathematical based model, advantages that many other engineering disciplines have exploited also. They can, also, greatly increase and enlarge our view about the system by revealing those inconsistencies, ambiguities and incompleteness we saw above. They are not meant to assure from start system correctness but to improve it and to reduce the probability of misinterpretation. Verification and testing are also widely recognized as an essential component of the full lifecycle of a complex system. While the former is concerned with the development of automatic or semiautomatic techniques that allow error detection and discovery of incoherences in a software system, testing on the other hand, is "the assessment by means of a test experiment that a product (implementation) conforms to its specification"[Tre92]. Otherwise said, verification is used to provide a guarantee that the system specification is correct, in the sense that it will properly work in all circumstances. It consists of formally proving, by mathematical means, the correctness of a formal description. Testing consists in checking that the interactions of the implementation with its environment trough some points, where it can be controlled and observed, conform to its specification. The implementation that is tested is stimulated by the environment which is represented here as an abstract entity called *tester*.

In the following we will talk about the existing methods for test derivation and verification such as *Labeled transition systems (LTS)* and *Input/Output Finite State Machines (I/O FSMs)*, the conformance test and its theoretical base, followed by some languages and tools which are already developed using the above principles. We will talk also about different methodologies that have emerged during time in order to improve performance or quality and to cope with the increased complexity. Statistically, each time a methodology or tool would be powerful enough to cope with present problems, people tend to extend in complexity or try to accommodate within a new problem for which the degree of complexity or manageability is increasing by an important amount in order of magnitude.

2 Testing and tests suites generation

Over the last twenty years a considerable amount of work has been done on the subject of automated generation of test suites. Nevertheless, their use in the real world is still in infancy[LL95] and this takes place because of the lack on their integration within assisted methodologies. Some of the automatic testing methods come from circuit testing and are based, as we already said, on automata theory[Gon70, Cho78a, SD88, VCI93, FBK+91, CKP93]. They consider the specification and the implementation as Mealy machines, as we will see in the following. These finite state automatons have transitions labeled with inputs and outputs. The general principle is to test each transition of the specification by starting in the initial state, applying inputs and finally check that the output is correct together with the fact that the finale state is the expected one. They essentially differ on the way the target is checked. The applicability of these methods is not always the more optimal one as they impose sometimes unrealistic assumptions on the specification. The algorithms can be very complex and sophisticated and can produce tests cases that are too long, but they are still used in today research as they did not have the last word yet.

Some other methods come from the testing theory [NH84, Abr87, Bri88, CKP93, Tre92].

Here the formalism used to model the implementation, the specification and the tests suites are labeled transition systems. This means that we have the normal states and the transitions are labeled with the actions that take place between states. A *conformance relation*, as we will see, defines which implementation is correct with respect to the specification. It allows to decide the conformance between the two of them[DAV93]. There can be a distinction between the input and output signals or not. Usually the outputs are only observable by the environment. They cannot be controlled from the point of view of the conformance relations[Tre95, Tre96]. A problem that arises here is the *test selection* as the tester has infinite behavior but testing activity must have a termination and a result, it cannot be infinite in time[FJJV96].

2.1 Testing review

Testing is the process of trying to find errors in a system by means of experimentation, which is usually carried out in a special environment, where normal and exceptional use is simulated for the given system. The aim of testing is to gain confidence that the system will work as *wanted* during normal use. Since the comportment of a real system is usually very complex and tends to have an infinite potential of manifesting and interacting within itself and with the exterior, testing cannot assure a complete correctness of an implementation. Testing can be done exhaustively only for trivial systems and this kind of testing process does not have interest anymore as it cannot be used often.

In order to realize the *testing process*, tests are applied to an implementation by an external entity called *tester* and the test outcomes are compared with the expected or calculated ones. Based on the results of the comparison, a verdict is formulated about the correctness of the implementation. Today, testing is applied mostly in the field of communication protocols as they represent a large class of software that can be modeled by finite state machines.

In software testing we often distinguish between *functional testing (black-box testing)* and *structural testing (white-box testing)*. We will present the two of them in the sections that concern conformance testing.

2.2 Test Architectures

The *test architecture* represents the description of the environment in which the test is performed, where the component or system under test is actually tested. It describes the embedding of the component in other systems during the testing process and how it communicates with the tester. A *test architecture*, conform to [ISO91], consists of :

- A tester;
- An Implementation under test(IUT);
- A test context;
- Some points of control and observation ("conformance points" or PCOs according to Open Distributed Processing (ODP) nomenclature[IT95]);
- Implementation access points or interfaces (IAPs);

The tester interacts with the *System Under Test* that is composed of the IUT and its context, interactions that can be observed and controlled in the *Points of Control and Observation (PCO)* that can be modeled, respectively, as queues:

- Input queue - for the observation of the events received from the IUT;
- Output queue - to control the events to be sent to the IUT;

In order to manage complexity issues that appear, a non-trivial system can be viewed as:

- A non-empty set of components under test (IUTs);
- A possibly empty set of components that are not concerned by testing(context);

When the testing context is empty we call the problem: *testing in isolation*. Here the interfaces and the points of observation coincide as opposite to the case where the context is not empty where we call the problem: *embedded testing*[LIM98].

One test is referred here as a *test case* and a *test suite* is a set of test cases that test all the conformance requirements. The tester behaves accordingly to the test suite that is obtained from a specification or reference system. The actual behaviour of the IUT is compared against the permitted behavior specified in the reference system or in the specification to which the conformance relation must be checked against.

In order to detect non-conformance, some transitions in the test cases, or an entire test case is associated with one of the following verdicts[FJJV96, LIM98]:

- **FAIL** means that the IUT does not conform to the specification. According to the conformance relation, a FAIL verdict is assigned to each input of the tester to whom output that does not correspond to any output of the specification.
- **PASS** means that the sequence from the initial state corresponds to an interaction sequence of the specification and is accepted by the test purpose automaton. This is the result that tells us that the IUT is conformant to its specification or to the respective aspect checked by the test case passed.
- **INCONCLUSIVE** means that there is insufficient evidence for the test to either PASS or FAIL. The behavior is, however, a valid one. It can be, also, a behavior that is not taken in account because of the fact that the test cannot be exhaustive, for example. It was a conforming one, but did not fulfil the requirements of the test purposes[Kni93].

A generic test architecture conform to [ISO91] is depicted in the figure 1.

Besides this generic architecture that describes the general approach and the general view of a testing environment, in practice [ISO91] defines four basic test architectures:

- ***Local Test Architecture*** - where the tester and the IUT are integrated together;
- ***Distributed Test Architecture*** - where the testers and IUT are external to each other;
- ***Coordinated Test Architecture*** - IUT and tester separated. They can be directly connected however;
- ***Remote Test Architecture*** - the tester and IUT are external to each other and the tester can be remotely connected to the IUT by means of a network;

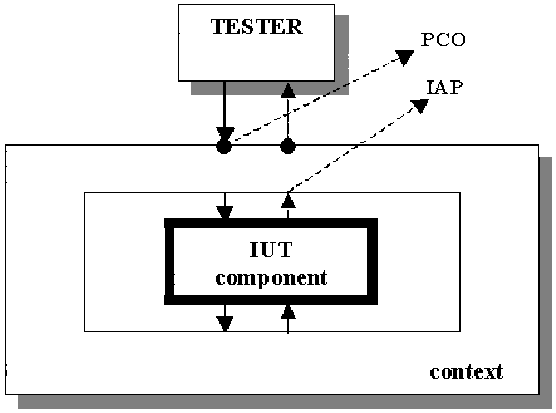

Figure 1: The Generic Test Architecture

2.3 Conformance testing

In practice there is a difference between the conformance test and system validation. Their goals can be easily confused. The difference between them lies in the entities they refer. One of them regards only the correctness of the specification while the other one establishes the relation of conformance between an implementation and its specification. To summarize:

- A *conformance test* is used to check that the external behavior of a given implementation of a system is equivalent to its formal specification;
- A *validation* is used to check that the formal specification itself is logically consistent;

If a formal specification has a design error, a faithful implementation of that specification should *pass* a conformance test if and only if it contains the same error. A conformance test should fail only if implementation and specification differ. On the other side, a consistency validation of a system should always reveal a design error. A conformance testing process can be viewed as the one presented in the figure 2.

Figure 2: The conformance testing process

Here the implementation is viewed as a black box with a finite set of inputs and outputs. The only type of interactions we can have with it is to provide it with sequences of input signals and to observe the resulting outputs. A series of input sequences, in this case, that is used to exercise the protocol implementation is called in this context a *conformance test suite*. The test is usually derived from the reference specification by a mechanical procedure. There are two mains problems to be solved:

- Finding a generally applicable, efficient procedure for generating conformance test suite given an implementation;

- Finding a method for applying the test suite to a running implementation;

The second problem may look simpler than it is, but the IUT might be a single layer or a component surrounded by other layers (2 or more) with interfaces to these layers. We can refer to this as the embedded test problem[LIM98]. Another problem is that the tester and the IUT can be physically separated from each other, as we saw in the case of remote testing architecture. This case is illustrated in the figure 3. We present in this figure the typical case of a network layer of a protocol (the IUT) that is tested over a network. The communication channel between the tester and the IUT is a virtual one whilst the actual communication is done by means of an actual physical channel that communicates, most of the time, indirectly with the IUT.

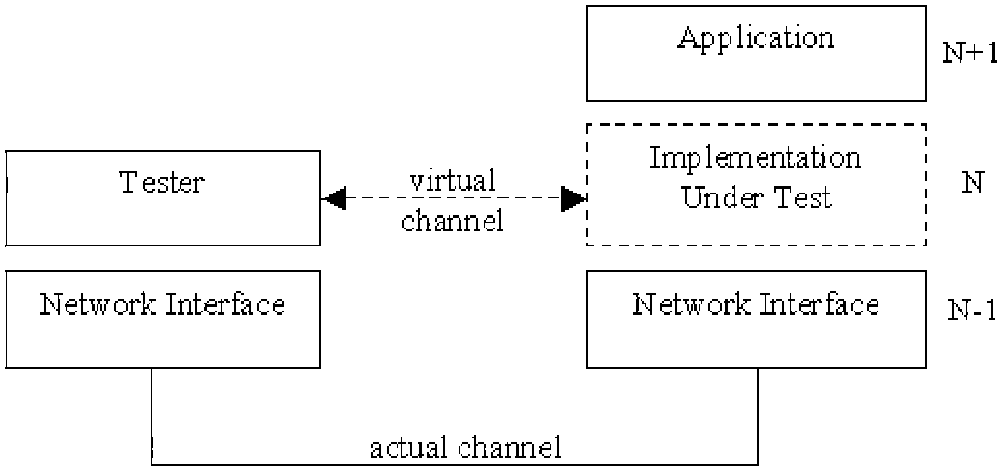

Figure 3: The general conformance problem

Another problem here is that the tester might not be able to supply inputs and retrieve outputs form the IUT in a completely reliable manner.

2.4 Functional Testing

In the early beginnings conformance testing became an issue when the administrators of the first public data networks had to determine the adequacy of commercial equipment that was used on their networks. The problem was to verify the conformance of the equipment to the network standard without having access to the proprietary, internal details of the equipment. At that time the first attempts to build effective test suites had two main goals:

- To establish that an implementation realizes all the functions of the original specification;
- To establish that an implementation can properly reject erroneous inputs in a way that is consistent with the original specification;

There are basically two types of faults detected by methods based on I/O FSMs. An output fault occurs when a transition produces an unexpected output for a given input. A transfer fault occurs when a transition leaves the implementation in an incorrect state (with respect to the specification).

In [Pa96] a fault model is defined as a triple $< A, \sim, R >$, where A is a reference I/O FSM (that of the specification), R, called the *fault domain*, is the set of all (possibly faulty)

implementation I/O FSMs defined over the same input alphabet X as the reference machine, and $\sim$ is a conformance relation. If the reference I/O FSM A is a deterministic minimal machine with n states, R_n is a set of all deterministic I/O FSMs over the same input alphabet as A with at most n states and $\sim$ is the trace equivalence relation $\cong$ then the fault model $< A, \cong, R_n >$ is the classical *black box* model, i.e. the one that corresponds to the functional testing. A complete test suite TS with respect to the fault model $< A, \cong, R_n >$ is a finite set of finite input sequences of the reference I/O FSM A such that for any FSM $B \epsilon R_n$, if $B \neq A$ then there exists an input sequence $\alpha \epsilon TS$ such that A and B produce different output sequences to α[LIM98].

2.5 Structural Testing

In this test we are not considering any data parameter values. Instead, the emphasis is on the control structure of the protocol. Trying to prove that the internal structure of a component is equivalent with the specification one is generally impossible as the number of tests to made can be infinite. It could be necessary to observe all predicted traces what is, except for the most trivial systems, impossible in a finite or, just reasonable, amount of time. In practice we have to make some simplifying assumptions[Hol91, LIM98]:

1. The IUT models a deterministic finite state machine with a known maximum number of states and with a known input and output vocabulary. Otherwise we could construct a machine which would pass a given test sequence by using as many states as there are transitions in the sequence. The practice in the literature is to consider that the implementation has no more states than the specification[CA95];

2. The specification I/O FSM is strongly connected, so that all states can be visited;

3. The specification I/O FSM is minimized. If not there are algorithms that can perform minimization for a given automaton;

In addition to these properties we can introduce others in order to manage different problems that can appear. One of them could be [Hol91]:

4. *(Completeness assumption)* - In each state the IUT can accept and respond to all input symbols from the complete system vocabulary; a null response i.e. a transition back to the same state is a valid response;

In [LIM98], the additional assumptions are grouped according to the following categories:

- *Uniformity hypotheses* state messages that contain parameters need only be tested with a limited (finite) number of instantiations (parameter values). We can consider most instances as being equivalent with regard to testing, according to this criterion;

- *Independency hypotheses* allow us to assume that a specification state that is attainable by several different paths remains unchanged in the implementation even if another path is chosen;

Another grouping criteria is present in [LIM98] that is considered as a requirement in [Hol91]:

- *Regularity hypotheses* allow us to fix an upper bound over the number of states of the implementation;

If we are to consider more simplifying, supplementary requirements, the IUT can have another three properties. They are convenient but not essential, however, as all they do is simplifying the task of conformance testing:

5. *Status property* - When a status message is received, the IUT responds with an output message that uniquely identifies the current state. The IUT does not change state;

6. *Reset property* - When a reset message is received, the IUT responds by making a transition to a known initial state, independent of its current state. The IUT need not produce an output;

7. *Set property* - When a set message is received in the initial system state, the IUT responds by making a transition to the state that is specified in a parameter of that message. The IUT need not produce an output;

2.6 The basic method

Given a machine with all these properties the conformance test can be performed as presented in the following algorithm:

Algorithm 1 - conformance testing

1. For all possible combinations of a state i and an input signal j, perform the following three steps.

2. Use the *reset* message to bring the IUT to the initial state, and then use the set message to transfer the IUT to the state *i*.

3. Apply input signal *j*. Verify that any output received, including the null output, matches the output required by the specification.

4. Use the *status* message to interrogate the IUT about its final state. Verify that this final state matches the one required by the specification.

The test verifies that the IUT is capable to correctly perform all state transitions in the formal specification. The set of input signals should include, in this case, the *set*, *reset* and *status* messages. If the IUT passes these tests, it is capable of reproducing the behavior of the formal specification but we cannot say anything more than that about its behavior.

Within the presented constraints this algorithm is the best we can hope to achieve with a conformance test. It remains to establish if it is the best possible one. The answer is intuitively: *no*. The cost of the test can be expressed as the length of the test suite, that is the total number of messages that is sent to the IUT. If we assume that the formal specification contains $S =\mid Q \mid$(the cardinality of the Q set) states and has an input vocabulary of V distinct messages, which includes the three additional messages we used the length of the test suite for this algorithm is $4SV$.

The following example shows how a test can be performed on a device represented by the automaton in figure 4. The generated test will start with the initial state (possibly any state) and will feed the IUT with messages (for each possible combination of an input message and a state). The *set message* takes one parameter: the target state. For the input column an s message followed by a parameter equates to the *set message* and the new state. The q message stands for the *status message*. A trace execution could look like the one presented in the table 1(the initial state would be S0 in this case).

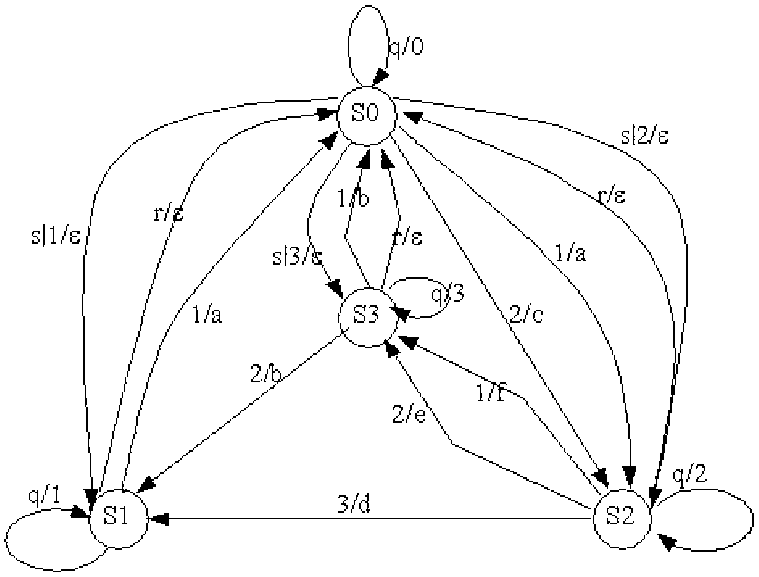

Figure 4: Testing using the basic method

No.crt	Input	Output	Crt. state		No.crt	Input	Output	Crt. state
1	1	a	S2		16	s|2	ε	S2
2	q	2	S2		17	1	f	S3
3	r	ε	S0		18	q	3	S3
4	s|0	ε	S0		19	r	ε	S0
5	2	c	S2		20	s|2	ε	S2
6	q	2	S2		21	3	d	S1
7	r	0	S0		22	q	1	S1
8	s|1	ε	S1		23	r	ε	S0
9	1	a	S0		24	s|3	ε	S3
10	q	0	S0		25	1	b	S0
11	r	ε	S0		26	q	0	S0
12	s|2	ε	S2		27	r	ε	S0
13	2	e	S3		28	s|3	ε	S3
14	q	3	S3		29	2	b	S1
15	r	ε	S0		30	q	1	S1

Table 1: Testing using the basic method

2.7 Transition tour

In the above algorithm after every test the IUT is forced back into the initial state. We can avoid that if we can find a sequence of state transitions that passes trough every state and every transition at least once. Such a sequence is called a *transition tour*. At best such a transition tour starts with a single reset message and exercises every transition exactly once, each time followed by a *status* message to verify the destination state. A set message is no longer required and the length of the test suite becomes $1 + 2SV$. The sufficient condition for the existence of an *Euler tour* (the equivalent in the graph theory) is expressed in the 2^{nd} requirement we have established together with the supplementary assumption that the graph is *symmetric* - i.e. every vertex (state) must be the destination and the origin of the same number of edges (transitions). In some of the dedicated literature the transition tour algorithm does not necessarily check for the validity of the final state of a transition. There are better algorithms, we will present in the following, that take in account more variables than the transition tour, and, that obtain better results.

As we saw, in the basic *algorithm 1* we needed all the three properties for it to work: the *reset*, *status* and *set*. The set message is an oddity that is not likely to be present in many IUTs, but fortunately it is not needed in any algorithm starting with the *transition tour*. The reset message can be replaced by a sequence of transitions, called a *homing sequence*, that is known to bring the system back to the initial state, whatever the current state might be. Such a sequence is constructed by an adaptive procedure where the answers of the FSM can be used in order to determine the next input message. It can be shown that every strongly connected finite state machine must have such an adaptive homing sequence, and, more than that, it can be algorithmically determined. It can be, also, shown that, in practice, we never need more than $S(S-1)/2$ transitions in order to get the machine in a known state[Moo56, Huf64]. To reach the initial system from here it takes between zero and $S-1$ extra transitions.

One transition tour we can derive for the automaton in the figure 5, having as starting node the S3 node, is: 1 3 1 2 1 2 3 2 2 2. This is the input sequence that must be feed to the automaton in order to obtain a valid transition tour. We should note that there are more than one possible transitions tours for an automaton.The problem, here, is that if we don't have a status message that could be applied after each transition we cannot verify the final state of the automaton.

2.8 Distinguishing sequences

We call a *distinguishing sequence* a sequence α (DS) of the FSM A for which the output responses of the IO/FSM to α at different states are different. It is a sequence that uniquely identifies each state of a I/O FSM. In other words when a I/O FSM A has a distinguishing sequence α, then the set $\{\alpha\}$ is a state identifier of any state of A. This mechanism can be used for the FSMs where we don't have a status message[Moo56, Koh78]. This is an important issue as it is not likely for a machine to have a message status if it is not required for the normal operation of the system. Compared to a homing sequence, a distinguishing sequence has the opposite goal: that of revealing the initial state of a transition sequence, instead of the last one. This is a general case for what we call an *Unique Input/Output* (UIO) sequence, which we will study in the next few sections. The process of deriving a complete test sequence in this context has two phases. In the first one the implementation is forced to display the responses of each state to the distinguishing sequence whilst in the second one, the verification part, an input is applied which causes the desired transition to be performed. Then the new state is identified by means of the distinguishing sequence. This method tests each transition, verifying after that

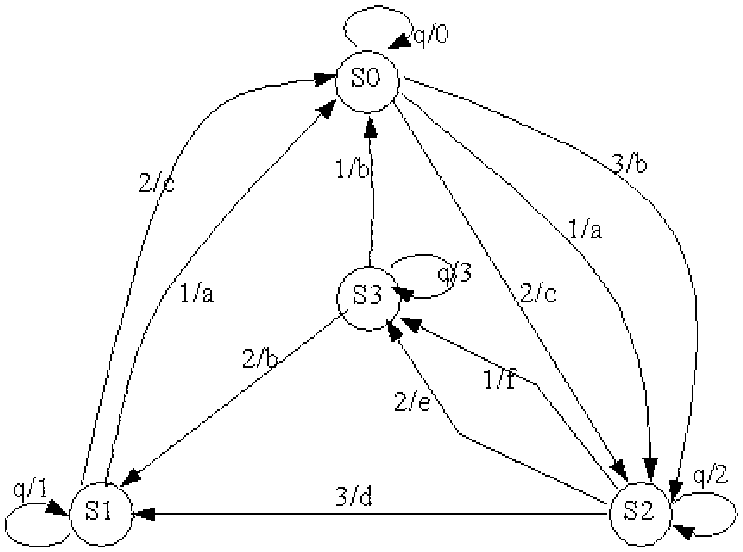

Figure 5: The transition tour

the final state with the determined distinguishing sequence. An example of a distinguishing sequence and the generated output is shown in figure 6 and table 2.

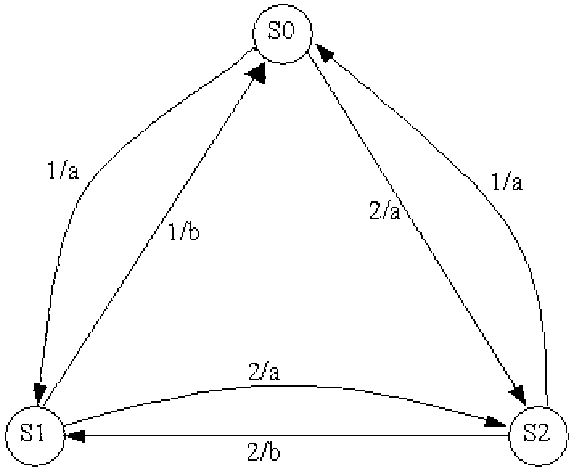

Figure 6: Distinguishing sequence

2.9 The W-method

This method appeared in order to resolve a problem with the distinguishing sequences: there are many FSMs that don't have such a distinguishing sequence. It can be shown, however, that for every minimized FSM there exists a finite set of input sequences such that the generated output sequences are unique for each state. The method that uses this issue is known by the *W-method* and is presented in theory with reset messages or without reset messages as shown in [Cho78b], respectively in [Hen64]. The key here is to replace the non-existent DS by sequences from the W set. The first part in deriving a test suite, here, is quite similar to that of the DS method except that each state has to display output for all sequences in the W set. As a

State	Output for DS=1.1	Preamble
S0	a.b	Null
S1	b.a	1/a
S2	a.a	2/a

Table 2: The distinguishing sequence for the automaton in figure 6

result this method often produces very long test sequences, this being a reason for the critiques that have been raised against it. The test consists in a preamble for each state followed by the transition to verify and the verification of the final state using the W set. An example for this method is depicted in the figure 7 and table 3.

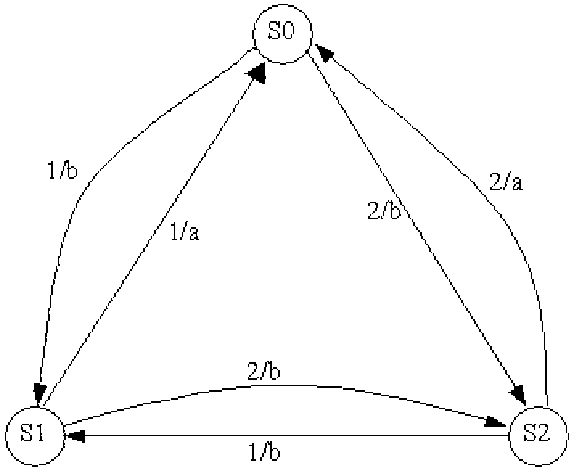

Figure 7: The W Method

State	Output for W={1,2}	Preamble
S0	{b,b}	Null
S1	{a,b}	1/b
S2	{b,a}	2/b

Table 3: The W set, preambles and outputs for the W method using the automaton in figure 7

2.10 The UIO (Unique Input Output) method

This methods works also for systems where the status message is missing by replacing it with an UIO sequence. The method overcomes the cases where a FSM does not have a distinguishing sequence also. An UIO sequence for a state s_i is an input/output sequence having as origin s_i such that there is no $s_j \neq s_i$ for which the UIO sequence is the same. The problem is more relaxed here as in the case of the distinguishing sequences as each UIO sequence can be different and, in fact, each state can have more than one UIO. Most FSMs have UIO sequences for every state, although there exists machines that don't have such sequences. They aren't

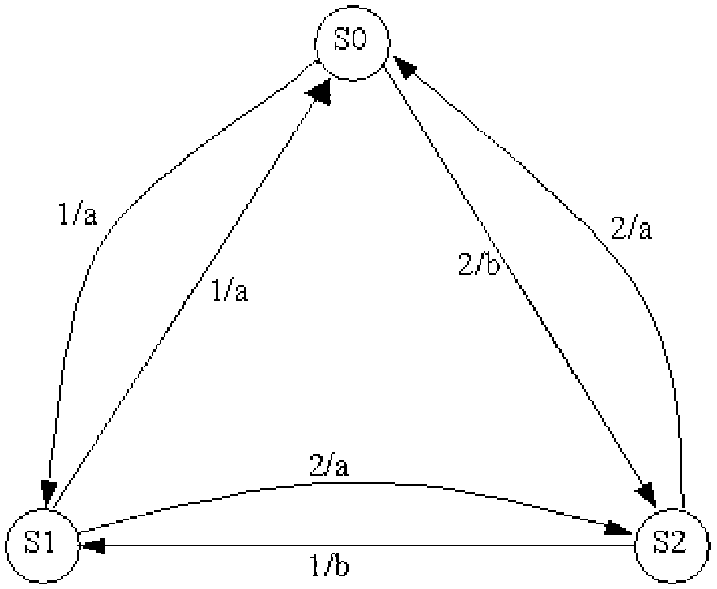

Figure 8: The UIO Method

usually longer than the distinguishing sequences. We can derive UIO sequences for I/O FSMs algorithmically. One of these algorithms is presented in [Hol91].

A problem that concerns the use of UIOs is the fact that they alter the current state of the FSM. They will not keep, in general, the machines in the same state as before applying the UIO, interfering thus with the testing algorithm used. This inconvenient can be avoided as shown in [Hol91] by adding in the graph of the original specification of a *pseudo-transition* for each input symbol of the state that must be checked. Another problem with the UIO method is that UIO sequences can only identify states in a correct IUT. If the IUT is faulty its behaviour is unpredictable. In order to solve this problem another approach can be used. It makes further assumptions about the types of faults in the IUT. The main assumption is that no fault can increase the number of reachable states or the number of input signals of the IUT. This method is based on *characterizing sequences* - a different finite sequence of inputs for every two states of a machine that triggers a different sequence of outputs[Hol91, Koh78].

For systems that have states that don't have UIOs a special method can be used to identify them. This is the *Partial UIO method*. Again, we have some optimizations of the UIO method, focused on the transition part, ignoring the state verification part. If used as proposed, without the corresponding state verification, none of them will provide a full fault coverage. These optimizations are shortly presented in the following sections.

As an example the UIO sequences for the automaton presented in the figure 8 are depicted in the table 4.

State	UIO(Si)	Preamble
S0	2/b	Null
S1	1/a 2/b	1/a
S2	1/b	2/b

Table 4: UIO sequences for the automaton in figure 8

2.10.1 Rural Chinese Postman Optimization (RCP Optimization)

This method consists in optimizing the cost of connecting the test segments by using transfer sequences that use not only reset and preamble sequences, but could include any specified transition. This alleviates the increased overhead cost in the non-optimized UIO version where each test segment is preceded by a reset followed by a preamble sequences having as purpose to put the implementation into the state $Head(tr)$ in order for the tr transition to be executed and tested. This problem is known also as the *Rural Chinese Postman Tour Problem.*

2.10.2 RCP with multiple UIOs

Another optimization is proposed by Lombardi in [LS92] that proposes to use different UIOs for identifying a state in different test segments. According his theory this can reduce the total length of the transition part. The idea is to obtain a graph closer to the symmetric one so that fewer edges need to be added in order to make it symmetric. The inconvenient of this method is that we can lose any gain we make in the transition verification by verifying uniqueness of the UIO sequences for multiple UIOs used in different test segments[LIM98].

2.10.3 RCP with overlaps

This optimization is based on the idea to eliminate the execution duplication of test segments that overlap. If we have two test segments for which the last part of one of them T_i coincides with the first part of the other one T_j, they can be merged so that the common part serve to both of them. The same process can be applied if one test segment is contained into another, in which case the former can be eliminated from the test suite. This was first time done by adding overlap links with negative costs to the graph[CCK90].

2.10.4 Greedy Overlap

Is another optimization method that uses a greedy algorithm to construct step by step the test sequences for the transition verification part instead of globally minimizing the transfer sequences between the segments.

2.10.5 Partial UIOs

This optimization is used for the FSMs that don't have UIO sequences for all states. The method is based on the *characterizing sequences* (we already presented) or *signatures* for every state that does not have UIOs. To verify a state that does not have an UIO, the signature is used. The signature for such a state is composed of $S-1$ sub-sequences (S is the number of states of the machine). Though there are $S(S-1)/2$ distinct pairs of states in a machine we never need more than $S-1$ of them in order to separate any combination of two states. This is proven in a constructive way by the following algorithm (Algorithm 2) that calculates signatures for every pair of different states.

Algorithm 2 - selecting characterizing sequences

1. Select two arbitrary states from the machine and find a sequence that separates them. The different output sequences in response to this sequence can be used to partition the S states into at least two different sets. The sets are blocks in a partitioning of states.

2. Select one of these blocks containing more than one state. Select two states from that block and find a sequencefrom the original collection that can separate them.

3. The number of state sets (blocks in a partitioning of the states) is increased by at least one extra set for each new characterizing sequence that we find. The procedure, therefore, can be repeated at most *S-1* times. At this time each block in the partitioning contains just a single state. For any two states in two different blocks we have now selected a sequence that can separate them. The set of *S-1* characterizing sequences selected from the original collection can be used to distinguish between any pair of states in the machine.

3 Testing in different environments

In the following we will present the testing process and some open issues in different environments. The first of them is the protocol testing area, followed by the open distributed object oriented testing area and the embedded testing platform. These categories should not be seen as independent ones, but as being in a very strong relation one with each other. In fact they overlap for the most of the real problems that are modeled.

3.1 Protocol testing

The problem of testing in the protocol area is maybe the oldest one. It started with the desire to design efficient error prone and unambiguous protocols. This desire existed long even before computer appeared and is represented by the different communication systems that were realized in the early beginnings of the communication revolution, from the fire signals to electrical ones and light signals. Successes and faults, both helped improving the idea about how such a communicating protocol must be efficiently designed. All the methods we presented in this paper have an entry point in the early theory of protocol testing and almost all the presented methods with their optimizations work with it. We can consider that protocol testing is a base for the testing research area, as this can be thought of in our days. Testing derived from here to the other methodologies and environments that emerged together with the networks, distributed systems, object-oriented techniques, etc.

3.2 Testing in Open Distributed OO Architectures

We speak here about open systems that are thought to be the systems that do not impose limitations to developers on particular hardware platforms, networks or software systems. Standards were deployed in order to assure the proposed aims of such systems. Some of these standards and reference models are: the *Open Distributed Processing (ODP)*, the *Common Object Request Broker (CORBA)* from Object Management Group, etc. The principal goals they propose are:

- A high level of interoperability between ODP systems concerning the information exchange, functionality use throughout the distributed system;
- Portability of applications across heterogeneous platforms;
- Different levels of transparency: data transparency, application transparency, data + application transparency;

Object oriented paradigm uses as low level notions only objects and messages. One is bound to using these primitives when working in such environment. Distributed systems have as special primitive or characteristic the message exchanging feature, which is made at higher or lower level of transparency. We can see that there is a link between the two paradigms: the message exchanging. As each paradigm offers its advantages today developers tend to express their

applications in terms of both paradigms. A distributed object oriented system tries to take advantage of this link by organizing its structures into objects and making all communications over the network take the form of messages between objects.

We can see, for example, that in order to assure the portability goal, object oriented paradigm offers us modularity and encapsulation. These two characteristics both help and assure a higher level of portability. Interoperability is based on a convenient and meaningful way of exchanging and sharing data in a distributed system. The use of multiple interfaces that is present in the OO technology and by thinking entities - objects - as being composed of smaller objects helps better achieving interoperability. Hiding complexity from the users and, possibly sometimes, from developers is best done trough abstraction. The concept of interfaces in object oriented approach can provide a good organization level and cleanness. There are, however, some open problems we have to face when comes to testing objects:

- Which parts of an object oriented program need to be tested and in what order ?
- What influence does inheritance have on testing, and do inherited software modules have to be subject to a repeated test ?
- How can be object-oriented testing be automated ?
- What are the strategies for guiding testing that can be applied in the object-oriented approach as the conventional ones do not map to the decentralized nature of object oriented software? What strategies could be used to guide class and cluster testing ?
- How can an object be specified in regard to testing ?
- What is a "testable interface" ?
- How are generic classes tested ?

Today some of these problems have answers, partial or complete ones. The others are still under research. Specification in object oriented has already undergone a vast amount of research[Boo94, Ra91] and answers have come out while the problem of testing the inheritance is still incomplete and there is no final or important solution to this. Some organizations have began studying the problem of object oriented testing using, however, the pragmatic *white-box* approach [JKSZ94] or by proposing a list of characteristics an object oriented tool should provide[Hun92].

Regarding conformance testing the ODP specifies a reference model that a complete system specification must contain:

- The behavior of the object being standardized and the way this behavior must be achieved;
- A list of the primitive terms used in the specification when making the statements about behavior;
- A conformance statement indicating the conformance points; what implementations must do at these points and what information implementers must supply ?

Regarding the way of testing, ODP documents specify two types of testing:

- *Passive testing* - all behavior is originated by the system under test and recorded by the tester (used especially to test client objects);

- *Active testing* - behavior is originated and recorded by the tester (used to test server objects);

Starting from here the test procedure takes place as described in the following. The specification of the system under test is in the form of an interface (the interface specification includes its behavior in ODP). The same is true for the tester and test procedures. During testing these interfaces are bound together. The tester must interpret its recorded observations using mapping provided by the implementor to yield propositions about the implementation which can then be checked to show that they are also true in the specification.

3.2.1 ODP approach to conformance

The object oriented paradigm imposes a complex abstract world. It is about the object complexity we are talking about here. Each object is usually composed of many other smaller objects which are in turn composed of other objects and so on. It is important here to define the level of abstraction where the testing procedure will take place. This will be the actual abstraction level of the specification. In ODP at any specific abstraction level, a test should be a series of observable activity: stimuli and events performed at *reference points* which are accessible interfaces. There are three types of basic interfaces:

1. **Signal Interface** - an interface in which all interactions are signals i.e. one-way atomic operations;
2. **Operational Interface** - an interface where all the interactions are operations;
3. **Stream interface** - an interface where all interactions take the form of information flows;

In the ODP model all system specifications must contain *reference points* and the respective behavior that must be satisfied at these points. This give place to the possibility of testing by means of some constraints of the real implementation[IT95]. These are the points that need to be available and accessible for test, in the implementation. To enlarge the definition of tests in object oriented systems - they are a series of observable stimuli and events performed at *reference points*, and only here. According to ODP the reference points are classified in four categories:

- The *Programmatic Reference Point* - is a reference point at which a programmatic interface can be established to allow access to a function. A programmatic interface is an interface which is realized trough a programming language binding;
- The *Perceptual reference Point* - is a reference point where we have interactions between the system and the real world, physical world;
- The *Inter-working Reference Point* - is a reference point where an interface can be established to allow communication between two or more systems. A conformance requirement is stated, here, in terms of the exchange of information between the systems.(Ex: OSI standards work at this level);
- The *Interchange Reference Point* - is a reference point where we can introduce into the system an external physical storage medium;

A problem here with the abstractions levels is that the more abstract a specification is, the more difficult is to test it. This happens because, in order to determine whether an abstract statement of the specification is true or not in the implementation, lots of implementation specific interpretation is needed. As research continues in this area there are many open issues and future work to be done in the domain of testing distributed object oriented systems. Some of them are [JTC94]:

- Are there any features of the ODP architecture which lead to unnecessary complexity in the conformance testing process?
- ODP objects can have multiple interfaces, and assessment of conformance to the objects behavior may require correlation of tests at a number of interfaces;
- A methodology is required for testing the conformance of stream interfaces;
- The ODP concept of environment contract allows performance constraints to be placed on the implementations, There is a need to study the methodology for testing conformance to these performance requirements(QoS for example);
- Study is needed to determine if extensions to methodology are needed to cover testing of client behavior as the current methodology is concentrated on testing objects acting as servers, i.e. the events are generated by clients;

Another problem is interoperability. Regarding product interoperability it is not yet very well defined what a conformance testing should do. Testing in this direction is split in two parts: conformance testing in isolation of each product followed by the interoperability testing, whilst the two of them should be performed together for the domain to be completely covered.

Even if some of these questions and problems have already partial answers and solutions, most of them need further work and research to be done.

3.3 Testing for embedded systems

This method is known in the literature as the gray box testing method and applies to composite systems where we have some components that have already been tested and proved to work correctly and other components that we need to test. The problem it resolve is that we might not want to test the system as a whole but only the non-tested yet components. This might be impossible using traditional methods as there are cases where the interfaces to the components to be tested are hidden inside the system which is seen as a black box. We speak about a black box only in what concerns the components to be tested, while we may have access to the interfaces of the other components of the system. The typical case is depicted in [LIM98]. A such system could generally be viewed as a system composed of two modules or set of modules: A for which the behaviour is known and verified and B, the module to be tested as in figure 9.

As we presented the system the B's interface is not accessible. Therefore the signals sent to B may reach the environment after traversing A and the other way around. A can be thought as a sort of a filter module. As a result of this, system responses to environment need to be correctly interpreted in order to check B's correctness. Traditional methods turn out to be inadequate for a few reasons[LIM98]:

- Module B can neither be removed from the system in order to be tested in isolation, nor it can give access to its internal interfaces. Traditional methods are, therefore, obliged to derive test sequences for the system as a whole;

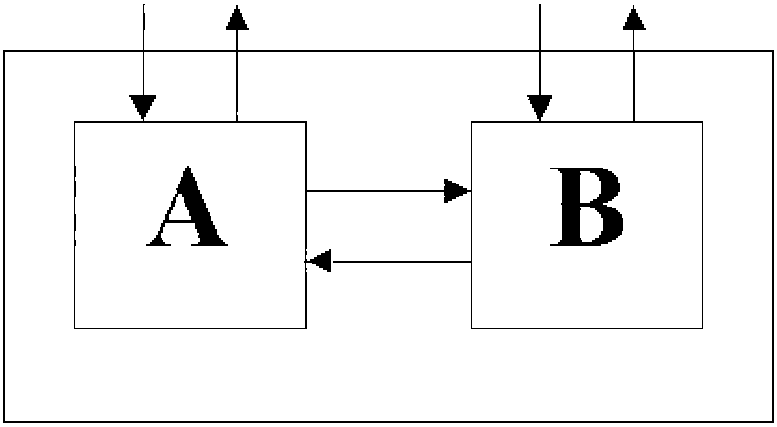

Figure 9: Embedded system testing

- Testing the system as a whole includes the testing of module B as well but we are testing unnecessarily, in the meantime, module A. This happens because the system's global behavior is likely to contain only behavior that concerns module A which does not need to be tested;

The key advantage of using embedded testing techniques is that they allow for the generation of test sequences concerning only the component under test, B in our case, avoiding the generation of redundant test sequences for the module A. Since the observable behavior of the system to be tested does not include internal messages between A and B, the external signals that we send must be chosen in such way so that they generate transitions in the module B. For this to happen, we must have details about the structure of the components within the system, about the configuration, even though we don't know anything about their implementation. Embedded system testing take, therefore, advantage of the available information about the configuration of the system components.

As we said before all these methodologies cannot be seen separately, but together, each one adding something to the others. The problem of embedded testing often occurs in functional testing of digital circuits and multiprocessor networks as well in testing object oriented packages which we treated in the previous section.

4 Tools used in test derivation and verification processes

In the following we briefly describe some tools that are currently used in deriving test sequences and in the verification process in our days. Some of these tools are developed inside research laboratories and are freely distributable.

4.1 Tools used in test derivation

TVeda is a test generation tool from CNET [Pha94] which accepts both Estelle and SDL formal description techniques and can produce test suites in TTCN format. TVeda-V3 can compute test kernels with two strategies: symbolic computation and reachability analysis. The specification is translated into an extended finite state machine (EFSM). TVeda computes a path from the source state of the EFSM to a target state in accordance with conditions on

variable values. The path is transformed into a tree when non-determinism of the specification is considered.

Topic V2 is a prototype of VERILOG developed inside the simulator of the commercial tool GEODE (see below) and based on a previous prototype: Topic. Its principles are quite similar to those of the work presented in [GDR93]. It takes as input an SDL specification and test purposes described as MSCs. The algorithm computes a graph constrained by the test purpose and unfolds it into a tree which is translated in TTCN.

TGV is a package developed in the CADP toolbox, presented below. GEODE is used for graph generation for the finite state machine from the SDL specification. Aldebaran is used afterwards for minimization of the transition system. TGV performs the test suite generation. It is based on verification techniques, synchronous product and *on the fly* verification.

4.2 Tools used in the verification process

GEODE is an SDL commercial tool developed by VERILOG. This tool supports requirement analysis, graphical design for data, architecture, communication and state machines, simulation and code generation. The simulator allows interactive simulation or automatic simulation for exhaustive verification.

CADP, the Caesar Aldebaran Distribution Package is a toolbox developed by Verimag at Grenoble. It is composed of several tools and environments among which Aldebaran and Open/Caesar.

Aldebaran is a tool which performs reduction and comparison of graphs according to various equivalence relations and procedures.

Open/Caesar is an open environment for rapid prototyping of verification algorithms or other algorithms based on traversals of transition systems. This is allowed by the presence of libraries for the management of graphs and memory. Transition systems may be given implicitly, by functions, by the LOTOS compiler, Caesar, or by other compilers, or, explicitly, in the Aldebaran graph format. The implicit representation is interesting for *on the fly* algorithms i.e. algorithms based on traversals of the transition system without explicit construction of the complete transition system.

SPIN is a verification tool developed at Bell laboratories. It permits for specification in a formal language: Promela. Verification is allowed in automatic mode and interactive mode. In order to solve the combinatorial explosion of the number of states that must be traversed it uses different strategies that decrease the amount of memory required. Some of them even eliminate the need for some kinds of problems to proceed to a exhaustive verification keeping an almost 100% coverage.

5 Conclusions

We have presented in this paper the problem of test suite generation for complex systems and the techniques used from the beginning to our days, with the different optimizations that have been applied to them during this time. We can see, that testing was the primary interest in the area of communication protocols where developers could not allow for a faulty protocol to be used in the real world. Many examples of faulty protocols are available[Hol91] and many of them have lead to important losses. Some of them are due to a faulty specification of the protocol while the others are due to the nonconformance of the implementation with the specification. Testing and verification can help in both case of errors. As the complexity increases the need for reliable and trustworthy systems increases also. This leads to an important amount of

research that is done in the direction of protocol testing and, lately, for open object oriented distributed systems testing. It still remains for the best results in this direction to show up, results and solutions that could impose themself in the academic and research world as well as in the industrial world.

The fact that an 100% proof that an implementation is correct in respect to a formal specification it is impracticable, maybe even impossible, the alternative is to develop test suites in such a way that they could give us a high level of confidence in the implementation we are testing. For the lower layers of the communication protocols a lot of studies and research has took place, in this direction, and results are showing up. Models have been applied and many test architectures have been developed. One of the major reasons for the orientation of the research towards the automatically generation of test suites is that the manual generation gives place to many errors that are not welcome at all in the process of testing and validation. Another reason is that test generation is by his means an iterative and repetitive process that suits very well for being automatically resolved.

Despite all these, only the first part of the road has been covered until now and a lot has been left for the future. Acceptation of this technology in the industrial world is not yet very clear. Almost all the studies that have been done are in the lower layers of communication protocols. Not very much has been done concerning the upper layers. More, these upper layers are more abstract. They are not as simple as the lower ones neither. As implementation is oriented, today, towards these protocol layers, appropriate validation and verification technologies must be developed.

The object orientation is another issue that has known a very rapid growth in the last years. Modeling techniques and languages like *UML (Unified Modeling Language)* have been developed in this direction. But after all this the testing and verification issues have been left behind in this area. In this approach the test deriving process has to be done in parallel with the design process. It is important here to supply methods and models that allow the integration of the testing issues directly with the design process.

The purpose of this work is oriented towards the test generation process for the higher layers of communicating protocols without leaving apart the object orientation. Test derivation using new methodologies and techniques is an important issue. Test derivation using on the fly verification techniques and accommodating tools like SPIN for test generation might be a good solution to a lot of today's problems, for example. Deriving test suites starting from UML formal descriptions, for object oriented systems, might be another possible way to follow. The research process in testing area will start to focus, without any doubt, on the higher level of abstraction imposed by today's systems and towards the higher layers of communicating protocols.

References

[Abr87] S. Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, (53(3)), 1987.

[Boo94] G. Booch. *Object Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, 2nd edition edition, 1994.

[Bri88] E. Brinskma. A theory for the derivation of tests. *Protocol Specification, Testing and Verification VIII*, pages 63–74, 1988.

[CA95] A. Cavalli and R. Anido. Verification and testing. *Eunice Summer School on Protocol Engineering*, 1995.

[CCK90] M.S. Chen, Y. Choi, and A. Kershenbaum. Approaches utilising segment overlap to minimise test sequences. *Proc. 10th international IFIP Symposium on Protocol Specification, Testing and Verification*, pages 67–84, 1990.

[Cho78a] T Chow. Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering*, (SE-4(3)), May 1978.

[Cho78b] T S Chow. Test software design modelled by finite state machines. *IEEE Transactions SE-4*, (3):178–187, 1978.

[CKP93] A.R. Cavalli, S.U. Kim, and Maigron P. Automated protocol conformance test generation based on formal methods for lotos specifications. In *Protocol Test Systems V*, pages 212–222. Elsevier Science Publisher B.V., Montreal, 1993.

[DAV93] K. Drira, P. Azema, and F. Vernadat. Refusal graphs for conformance tester generation and simplification. *Protocol Specification, Testing and Verification XIII*, 1993.

[FBK+91] S Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, (17(6)), june 1991.

[FJJV96] J.C. Fernandez, C. Jard, T. Jeron, and C. Viho. An experiment in automatic generation of tests suites for protocols with verification technology. Theme 1 No.0000, Institut National De Recherche en Informatique et Automatique, IRISA Rennes, May 1996.

[GDR93] J. Grabowski, D.Hogrefe, and R.Nahm. Test case generation with test purpose specification by mscs. *Elsevier Sceince B.V., 6th SDL Forum*, pages 253–266, 1993.

[Gon70] G. Gonenc. A method for the design of fault-detection experiments. *IEEE Transactions of Computing*, (C-19), June 1970.

[Hen64] F.C. Hennie. Fault detecting experiments for sequential circuits. In *Proceedings of 5th Ann. Symp. Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*, volume 1. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Huf64] D.A. Huffman. Canonical forms for information losseless finite state logical machines. *Sequential Machines:Selected Papers*, 1964.

[Hun92] B. Hunt. Practical object-oriented programming:38 guidelines for making oop work. *EDN Journal*, pages 138–146, July 1992.

[ISO91] ISO. Information technology, open systems interconnection, conformance testing methodology and framework. *Internationa Standards ISO-9646*, 1991.

[IT95] ISO/ITU-T. Open distributed processing -reference model - part 2: Foundations. *Internationa Standard 10746-2*, 1995.

[JKSZ94] P. Juttner, S. Kolb, S. Sieber, and P. Zimmerer. Testing major object-oriented software systems. *Siemens Review-RD*, (Special):25–29, 1994.

[JTC94] ISO/IEC JTC1/SC21/WG1. Revised working draft on formal methods in conformance testing. *Output form Southampton*, July 1994.

[Kni93] K. G. Knightson. *OSI Protocol Conformance Testing - IS9646 Explained.* McGraw-Hill, 1993.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* Computer Science Series. McGraw-Hill, New York, 2nd edition edition, 1978.

[LIM98] Luiz Augusto DE PAULA LIMA. *Une Méthode Pragmatique de Génération de Séquences de Test pour les Systèmes Imbriqués (et Applications aux Plates-Formes Ouvertes Distribuées et aux Services de Télécommunications).* PhD thesis, Université d'Evry - Val d'Essonne,Institut National des Télécommunications, Nov 1998.

[LL95] R Lai and W Leung. Industrial and academic protocol testing: the gaps and the means of convergence. *Computer Networks and ISDN Systems*, (27):537–547, 1995.

[LS92] F. Lombardi and Y.N. Shen. Evaluation and improvement of fault coverage of conformance testing by uio sequences. *IEEE Transactions on Communications*, 40(8):1288–1293, August 1992.

[Moo56] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematical Studies*, (34):129–153, 1956.

[NH84] R. De Nicola and M. Henessy. Testing equivalences for processes. *Theoretical Computer Science*, (34):83–133, 1984.

[Pa96] A. Petrenko and al. Testing in context: framework test derivation. *Computer Communications*, 19:1236–1249, 1996.

[Pha94] M. Phalippou. Test sequence using estelle or sdl structure information. *FORTE 94*, October 1994.

[Ra91] J Rumbaugh and al. *Object Oriented Modelling and Design.* Prentice Hall, 1991.

[SD88] K. Sabnani and A Dabhura. A protocol testing procedure. *Computer Networks and ISDN Systems*, (15(4)), 1988.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Trente, Netherlands, 1992.

[Tre95] J. Tretmans. Testing labelled transition systems with inputs and outputs. In *8th International Workshop on Protocols Test Systems*, Evry-France, September 1995.

[Tre96] J. Tretmans. Test generation with inputs, outputs and quiescence. In B. Steffen T. Margaria, editor, *Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'96, Passau, Germany, March 1996.

[VCI93] S Vuong, W Chan, and M Ito. The uiov method for protocol test sequence generation. *Second International Workshop on Protocol Test Systems*, 1993.